Formal Characterization of Illegal Control Flow in Android System

Mariem Graa, Nora Cuppens-Bouhlahia, Frédéric Cuppens, Ana Cavalli

Formal Characterization of Illegal Control Flow in Android System

Mariem Graa*†, Nora Cuppens-Boulahia*, Frédéric Cuppens*, and Ana Cavalli†
*Telecom-Bretagne, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné - France
Emails: {mariem.benabdallah,nora.cuppens,frederic.cuppens}@telecom-bretagne.eu
†Telecom-SudParis, 9 Rue Charles Fourier, 91000 Evry - France
Emails: {mariem.graa,ana.cavalli}@it-sudparis.eu

***Abstract*—The dynamic taint analysis mechanism is used to protect sensitive data in the Android system. But this technique does not detect control flows which can cause an under-tainting problem. This means that some values should be marked as tainted, but are not. The under-tainting problem can be the cause of a failure to detect a leak of sensitive information. To solve this problem, we use a set of formally defined rules that describes the taint propagation. We prove the completeness of these rules. Also, we provide a correct and complete algorithm based on these rules to solve the under-tainting problem.**

***Keywords*-dynamic taint analysis; android system; control flows; under-tainting; formal rules; complete algorithm**

I. Introduction

Recent years have witnessed an increase in the use of embedded systems such as smartphones. According to a recent Gartner report [1], 417 million of worldwide mobile phones were sold in the third quarter of 2010, which corresponds to 35 percent increase from the third quarter of 2009. To make mobile phones more fun and useful, users usually download third-party applications. For example, we can show an increase in third-party apps of Android Market from about 15,000 third-party apps in November 2009 to about 150,000 in November 2010. These applications are used to capture, store, manipulate, and access to data of a sensitive nature in mobile phone. An attacker can launch flow control attacks to compromise confidentiality and integrity of the Android system and can leak private information without user authorization. In the study presented in Black Hat conference, Daswani [2] analyzed the live behavior of 10,000 Android applications and show that more than 800 were found to be leaking personal data to an unauthorized server. Therefore, there is a need to provide adequate security mechanisms to control the manipulation of private data by third-party apps. The dynamic taint analysis mechanism is used to protect sensitive data in the Android system against attacks [3]. But this technique does not detect control flows which can cause an under-tainting problem i.e. that some values should be marked as tainted, but are not. Let us consider the attack shown in Figure 1 that presents an under-tainting problem which can cause a failure to detect a leak of sensitive information. The variables x and y are both initialized to false. On Line 4, the attacker tests the user's input for a specific value. Let us assume that the attacker was lucky and the test was positive. In this case, Line 5 is executed, setting x to true and x is tainted. Variable y keeps its false value, since the assignment on Line 7 is not executed and y is not tainted because dynamic tainting occurs only along the branch that is actually executed. As y is not tainted, it is leaked to the network (Line 8) without being detected. Since y has not been modified, it informs the attacker about the value of the user private contact. Thus, an attacker can circumvent an android system through the control flows.

```
1.x= false;
2.y=false;
3.char c[256];
4.if( gets(c) != user_contact )
5.      x=true;
6.else
7.      y=true;
8.NetworkTransfer (y);
```

Figure 1. Attack using indirect control dependency

In a previous work [4], we have proposed an approach that combines static and dynamic taint analysis to propagate taint along control dependencies and to track implicit flows in embedded systems such as the Google Android operating system. In this paper, we formally specify the under-tainting problem and we provide an algorithm to solve it based on a set of formally defined rules describing the taint propagation. We prove the completeness of those rules and the correctness and completeness of the algorithm.

The rest of this paper is organized as follows. Section 2 gives a technical overview of our approach. Section 3 presents some definitions and theorems that are used in other sections. Section 4 describes our formal specification of the under-tainting problem. In section 5, we specify an algorithm based on a set of formally defined rules describing the taint propagation policy that we use to solve the under-tainting problem. Related work about existing solutions to solve the under-tainting problem is analyzed in section 6. Finally, section 7 concludes with an outline of future work.

II. Approach Overview

TaintDroid [3], an extension of the Android mobile-phone platform, implements dynamic taint analysis to track the information flow in real-time and control the handling of private data. It addresses different challenges specific to mobile phones like the resource limitations. TaintDroid is composed of four modules: (1) Explicit flow module that tracks variable at the virtual machine level, (2) IPC Binder module that tracks messages between applications, (3) File module that tracks files at the storage level and (4) Taint propagation module that is implemented in the native methods level. It only tracks explicit flows and does not track control flows. In a previous work [4], we have proposed a technical approach that enhances the TaintDroid approach by tracking control flow in the Android system to solve the under-tainting problem. To track implicit flow, we have added an implicit flow module in the Dalvik VM bytecode verifier which checks instructions of methods at load time. We have defined two additional rules to propagate taint in the control flow. At class load time, we have built an array of variables to which a value is

assigned to handle the branch that is not executed. Figure 2 presents the modified architecture to handle implicit flow in TaintDroid system.

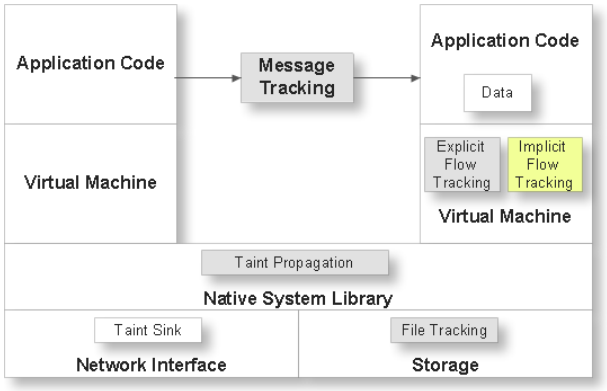

Figure 2. Modified architecture to handle implicit flow in TaintDroid system.

In this paper, we prove the completeness of the two additional rules and we provide a correct and complete taint algorithm based on these rules. Our algorithm is based on a hybrid approach that combines and benefits from the advantages of static and dynamic analyses. We use static analysis to detect control dependencies. This analysis is based on the control flow graphs [5], [6] which will be analyzed to determine branches in the conditional structure. A basic block is assigned to each control flow branch. Then, we detect the flow of the condition-dependencies from blocks in the graph. Also, we detect variable assignment in a basic block of the control flow graph to handle not executed branches. The dynamic analysis uses information provided by the static analysis and allows tainting variables to which a value is assigned in the conditional instruction. To taint these variables, we create an array of context taints that includes all condition taints. We use the context taints array and the condition-dependencies from block in the graph to set the context taint of each basic block. Finally, we apply the propagation rules to taint variables to which a value is assigned whether the branch is taken or not.

III. Notations, Definitions and Theorems

Definition 1. *Direct graph*
A directed graph $G = (V, E)$ consists of a finite set V of vertices and a set E of ordered pairs (v, w) of distinct vertices, called edges. If (v, w) is an edge, w is a successor of v and v is a predecessor of w.

Definition 2. *Complete directed graph*
A complete directed graph is a simple directed graph $G = (V, E)$ such that every pair of distinct vertices in G are connected by exactly one edge. So, for each pair of distinct vertices, either (x, y) or (y, x) (but not both) is in E.

Definition 3. *Control flow graph*
A control flow graph $G = (V, E, r)$ is a directed graph (V, E) with a distinguished $Exit$ vertex and start vertex r, such that for any vertex $v \in V$ there is a path from r to v. The nodes of the control flow graph represent basic blocks and the edges represent control flow paths.

The concept of post-dominator and dominator tree are used to determine dependencies of blocks in the control flow graph.

Definition 4. *Dominator*
A vertex v dominates another vertex $w \neq v$ in G if every path from r to w contains v.

Definition 5. *Post-Dominator*
A node v is post-dominated by a node w in G if every path from v to $Exit$ (not including v) contains w.

Theorem 1. Every vertex of a flow graph $G = (V, E, r)$ except r has a unique immediate dominator. The edges $\{(idom(w), w) | w \in V - \{r\}\}$ form a directed tree rooted at r, called the dominator tree of G, such that v dominates w if and only if v is a proper ancestor of w in the dominator tree[7], [8].

Computing post-dominators in the control flow graph is equivalent to computing dominators [5] in the reverse control flow graph. Dominators in the reverse graph can be computed quickly by using the Fast Algorithm [9] or a linear-time dominators algorithm [10] to construct the dominator tree. Using these algorithms, we can determine the post-dominator tree of a graph.

Definition 6. *Control Dependency* Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X noted $Dependency(X, Y)$ if:

1) There exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and

2) X is not post-dominated by Y.

Given the post-dominator tree, Ferrante *et al.* [11] determine control dependencies by examining certain control flow graph edges and annotating nodes on the corresponding tree paths.

Definition 7. *Context_Taint*
Let G be a control flow graph. Let X and Y be basic blocks in G. If Y is control dependent on X that contains $Condition$ then we associate to Y a $Context\_Taint$ with $Context\_Taint(Y) = Taint(Condition)$.

We use the completeness theorem to prove the completeness of the taint propagation rules in section V-A. We use the soundness theorem to prove this completeness from left to right and the compactness theorem and theorem 2 to prove from right to left. These theorems [12] are given below.

Completeness Theorem. For any sentence G and set of sentences $\mathcal{F}$, $\mathcal{F} \models G$ if and only if $\mathcal{F} \vdash G$.

Soundness Theorem. For any formula G and set of formulas $\mathcal{F}$, if $\mathcal{F} \vdash G$, then $\mathcal{F} \models G$.

Compactness Theorem. Let $\mathcal{F}$ be a set of formulas. $\mathcal{F}$ is unsatisfiable if and only if some finite subset of $\mathcal{F}$ is unsatisfiable.

Definition 8. *CNF formula*
A formula F is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals. That is,

$$F = \bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m} L_{i,j})$$

where each $L_{i,j}$ is either atomic or a negated atomic formula.

Theorem 2. Let F and G be formulas of the first-order logic. Let H be the CNF formula obtained by applying the CNF algorithm [12] to the formula $F \wedge \neg G$. Let $Res^*(H)$ be the set of all clauses that can be derived from H using resolvents. The following are equivalent:

1) $F \models G$
2) $F \vdash G$
3) $\emptyset \in Res^*(H)$

IV. The under-tainting Problem

In this section we formally specify the under-tainting problem based on Denning's information flow model. Denning [13] defined an information flow model as:

$$FM =< N, P, SC, \oplus, \rightarrow > .$$

N is a set of logical storage objects (files, program variables, ...). P is a set of processes that are executed by the active agents responsible for all information flow. SC is a set of security classes that are assigned to the objects in N. SC is finite and has a lower bound L attached to objects in N by default. The class combining operator "$\oplus$" specifies the class result of any binary function on values from the operand classes. A flow relation "$\rightarrow$" between pairs of security classes A and B means that "information in class A is permitted to flow into class B". A flow model FM is secure if and only if execution of a sequence of operations cannot produce a flow that violates the relation "$\rightarrow$".

We draw our inspiration from the Denning information flow model to formally specify under-tainting. However, we assign taint to the objects instead of assigning security classes. Thus, the class combining operator "$\oplus$" is used in our formal specification to combine taints of objects.

Syntactic definition of connectors $\{\Rightarrow, \rightarrow, \leftarrow, \oplus\}$:
We use the following syntax to formally specify under-tainting: A and B are two logical formulas and x and y are two variables.

- $A \Rightarrow B$: If A then B
- $x \rightarrow y$: Information flow from object x to object y
- $x \leftarrow y$: the value of y is assigned to x
- $Taint(x) \oplus Taint(y)$: specifies the taint result of combined taints.

Semantic definition of connectors $\{\rightarrow, \leftarrow, \oplus\}$:

- The $\rightarrow$ connector is reflexive: If x is a variable then $x \rightarrow x$.
- The $\rightarrow$ connector is transitive: x, y and z are three variables, if $(x \rightarrow y) \wedge (y \rightarrow z)$ then $x \rightarrow z$.
- The $\leftarrow$ connector is reflexive: If x is a variable then $x \leftarrow x$.
- The $\leftarrow$ connector is transitive: x, y and z are three variables, if $(x \leftarrow y) \wedge (y \leftarrow z)$ then $x \leftarrow z$.
- The $\rightarrow$ and $\leftarrow$ connectors are not symmetric.
- The $\oplus$ relation is commutative: $Taint(x) \oplus Taint(y) = Taint(y) \oplus Taint(x)$
- The $\oplus$ relation is associative: $Taint(x) \oplus (Taint(y) \oplus Taint(z)) = (Taint(x) \oplus Taint(y)) \oplus Taint(z)$

Definition 9. *Under-Tainting*
We have a situation of under-tainting when x depends on a *condition*, the value of x is assigned in the conditional branch and *condition* is tainted but x is not tainted.

Formally, an under-tainting occurs when there is a variable x and a formula *condition* such that:

$$\begin{aligned} &\textit{IsAssigned}(x, y) \wedge Dependency(x, condition) \\ &\wedge Tainted(condition) \wedge \neg Tainted(x) \end{aligned} \tag{1}$$

where:

- *IsAssigned*(x, y) associates with x the value of y.

$$\textit{IsAssigned}(x, y) \stackrel{def}{\equiv} \quad (x \leftarrow y)$$

- $Dependency(x, condition)$ defines an information flow from *condition* to x when x depends on the *condition*.

$$Dependency(x, condition) \stackrel{def}{\equiv} \quad (condition \rightarrow x)$$

V. The under-tainting Solution

In this section, we specify a set of formally defined rules that describe the taint propagation. We prove the completeness of these rules. Then, we provide an algorithm to solve the under-tainting problem based on these rules. Afterwards, we analyse some important properties of our algorithm such as Correctness and Completeness.

A. The taint propagation rules

Let us consider the following axioms:

$$(x \rightarrow y) \Rightarrow (Taint(y) \leftarrow Taint(x)) \tag{2}$$

$$(x \leftarrow y) \Rightarrow (y \rightarrow x) \tag{3}$$

$$\begin{aligned} &(Taint(x) \leftarrow Taint(y)) \wedge (Taint(x) \leftarrow Taint(z)) \\ &\Rightarrow (Taint(x) \leftarrow Taint(y) \oplus Taint(z)) \end{aligned} \tag{4}$$

Theorem 3. We consider that $Context\_Taint$ is the taint of the *condition*. To solve the under-tainting problem, we use the two rules that specify the propagation taint policy:

- Rule 1: if the value of x is modified and x depends on the *condition* and the branch is taken, we will apply the following rule to taint x.

$$\begin{aligned} &IsModified(x, explicitflowstatement) \\ &\wedge (Dependency(x, condition)) \\ &\wedge (BranchTaken(br, conditionalstatement)) \\ &\vdash (Taint(x) \leftarrow Context\_Taint \\ &\oplus Taint(explicitflowstatement)) \end{aligned}$$

where: The predicate $BranchTaken(br, conditionalstatement)$ specifies that branch br in the $conditionalstatement$ is executed. So, an explicit flow which contains x is executed. *IsModified (x, explicitflowstatement)* associates with x the result of an explicit flow statement.

$$\begin{aligned} &IsModified(x, explicitflowstatement) \stackrel{def}{\equiv} \\ &(x \leftarrow explicitflowstatement) \end{aligned}$$

- Rule 2: if the value of y is assigned to x and x depends on the *condition* and the branch br' in the conditional statement is not taken (x depends only on implicit flow and does not depend on explicit flow), we will apply the following rule to taint x.

$$IsAssigned(x,y) \wedge Dependency(x, condition)$$
$$\wedge \neg BranchTaken(br', conditionalstatement)$$
$$\vdash Taint(x) \leftarrow Taint(x) \oplus Context\_Taint$$

Proof of taint propagation rules

To prove completeness of propagation taint rules, we use the basic rules cited in Table I for derivations.

Premise	Conclusion	Name
G is in $\mathcal{F}$	$\mathcal{F} \vdash G$	Assumption
$\mathcal{F} \vdash G$ and $\mathcal{F} \subset \mathcal{F}'$	$\mathcal{F}' \vdash G$	Monotonicity
$\mathcal{F} \vdash F$,$\mathcal{F} \vdash G$	$\mathcal{F} \vdash (F \wedge G)$	$\wedge$-Introduction
$\mathcal{F} \vdash (F \wedge G)$	$\mathcal{F} \vdash (G \wedge F)$	$\wedge$-Symmetry

Table I
BASIC RULES FOR DERIVATIONS

We start by proving completeness of the first rule. We suppose that $\mathcal{F} = \{IsModified(x, explicitflowstatement), Dependency(x, condition), BranchTaken(br, conditionalstatement)\}$ and $G = Taint(x) \leftarrow Context\_Taint \oplus Taint(explicitflowstatement)$.

We prove soundness, left to right, by induction. If $\mathcal{F} \vdash G$, then there is a formal proof concluding with $\mathcal{F} \vdash G$ (see Table II). Let M be an arbitrary model of $\mathcal{F}$, we will demonstrate that $M \models G$. G is deduced by Modus ponens of $G_j, G_j \rightarrow G$ then by induction, $M \models G_j$ and $M \models G_j \rightarrow G$ and it follows $M \models G$.

Statement	Justification
1. $(condition \rightarrow x) \vdash (Taint(x) \leftarrow Taint(condition))$	Axiom (2)
2. $(condition \rightarrow x) \vdash (Taint(x) \leftarrow Context\_Taint)$	Taint(condition) = Context_Taint
3. $\mathcal{F} \vdash (Taint(x) \leftarrow ContextTaint)$	Monotonicity applied to 2
4. $(x \leftarrow explicitflowstatement) \vdash (explicitflowstatement \rightarrow x)$	Axiom (3)
5. $(x \leftarrow explicitflowstatement) \vdash (Taint(x) \leftarrow Taint(explicitflowstatement))$	Axiom (2)
6. $\mathcal{F} \vdash (Taint(x) \leftarrow Taint(explicitflowstatement))$	Monotonicity applied to 5
7. $\mathcal{F} \vdash ((Taint(x) \leftarrow Context\_Taint) \wedge (Taint(x) \leftarrow Taint(explicitflowstatement)))$	$\wedge$-Introduction applied to 3 and 6
8.$\mathcal{F} \vdash G$	Modus ponens

Table II
FORMAL PROOF OF THE FIRST RULE

Conversely, suppose that $\mathcal{F} \models G$, then $\mathcal{F} \cup \neg G$ is unsatisfiable. By compactness, some finite subset of $\mathcal{F} \cup \neg G$ is unsatisfiable. So there exists finite $\mathcal{F}_0 \subset \mathcal{F}$ such that $\mathcal{F}_0 \cup \neg G$ is unsatisfiable and, equivalently, $\mathcal{F}_0 \models G$. Since $\mathcal{F}_0$ is finite, we can apply Theorem 2 to get $\mathcal{F}_0 \vdash G$. Finally, $\mathcal{F} \vdash G$ by Monotonicity. ■

We will now prove completeness of the second rule. We assume again that $\mathcal{F} = \{IsAssigned(x,y), Dependency(x, condition), \neg BranchTaken(br', conditionalstatement)\}$ and $G = Taint(x) \leftarrow Taint(x) \oplus Context\_Taint$.

Similarly to the first rule, we prove soundness by induction. If $\mathcal{F} \vdash G$, then there is a formal proof concluding with $\mathcal{F} \vdash G$ (see Table III).

Let M be an arbitrary model of $\mathcal{F}$, we will demonstrate that $M \models G$. G is deduced by Modus ponens of $G_j, G_j \rightarrow G$ then by induction, $M \models G_j$ and $M \models G_j \rightarrow G$ and it follows $M \models G$.

Statement	Justification
1. $(condition \rightarrow x) \vdash (Taint(x) \leftarrow Taint(condition))$	Axiom (2)
2. $(condition \rightarrow x) \vdash (Taint(x) \leftarrow Context\_Taint)$	Taint(condition) = Context_Taint
3. $\mathcal{F} \vdash (Taint(x) \leftarrow Context\_Taint)$	Monotonicity applied to 2
4. $x \vdash (x \leftarrow x)$	The relation $\leftarrow$ is reflexive
5. $\mathcal{F} \vdash (x \leftarrow x)$	Monotonicity applied to 3
6. $(x \leftarrow x) \vdash (Taint(x) \leftarrow Taint(x))$	Axiom (2)
7. $\mathcal{F} \vdash (Taint(x) \leftarrow Taint(x))$	Modus ponens applied to 5 and 6
8. $\mathcal{F} \vdash ((Taint(x) \leftarrow Context\_Taint) \wedge (Taint(x) \leftarrow Taint(x)))$	$\wedge$-Introduction applied to 3 and 7
9. $\mathcal{F} \vdash ((Taint(x) \leftarrow Taint(x)) \wedge (Taint(x) \leftarrow Context\_Taint))$	$\wedge$-Symmetry applied to 8
10.$\mathcal{F} \vdash G$	Modus ponens

Table III
FORMAL PROOF OF THE SECOND RULE

Conversely, suppose that $\mathcal{F} \models G$. Then $\mathcal{F} \cup \neg G$ is unsatisfiable. By compactness, some finite subset of $\mathcal{F} \cup \neg G$ is unsatisfiable. So there exists finite $\mathcal{F}_0 \subset \mathcal{F}$ such that $\mathcal{F}_0 \cup \neg G$ is unsatisfiable and, equivalently, $\mathcal{F}_0 \models G$. Since $\mathcal{F}_0$ is finite, we can apply Theorem 2 to get $\mathcal{F}_0 \vdash G$. Finally, $\mathcal{F} \vdash G$ by Monotonicity. ■

B. The algorithm

The tainting algorithm that we propose, $Taint\_Algorithm$, allows solving the under-tainting problem. It takes as input a control flow graph of a binary program. In this graph, nodes represent a set of instructions that represent basic blocks. Firstly, it determines the control dependency of the different blocks in the graph using $Dependency\_Algorithm$ [11]. Afterwards, we parse the $Dependency\_List$ generated by $Dependency\_Algorithm$ and we set the context taint of blocks to include the taint of the condition that depends on whether the branch is taken or not. Finally, using the context taint and the two propagation rules, we taint all variables to which a value is assigned in the conditional branch.

Algorithm 1 Taint_Algorithm (Control flow graph G)

Input: $G = (V, E, r)$ is a control flow graph of a binary program
Output: $Tainted\_Variables\_List$ is the list of variables that are tainted.

$x \in V$
$y \in V$
$Dependency\_List \leftarrow Dependency\_Algorithm(G)$
while $(x, y) \in Dependency\_List$ **do**
  $Set\_Context\_Taint(y, List\_Context\_Taint)$
  $Tainted\_Variables\_List \leftarrow$
  $Taint\_Assigned\_Variable(y)$
end while

```
boolean x;
boolean y;
if ( x == true )
y = t rue;
else
y = false;
return(y);
```

```
0: iload_0
1: ifeq 9
4: iconst_1
5: istore_1
6: goto 11
9: iconst_0
10: istore_1
11: iload_1
12: ireturn
```

Figure 3. Source code example Figure 4. Bytecode example

C. Running example

We analyze the control flow graph $G = (V, E, r)$ (see Figure 5) of the bytecode given in Figure 4 to illustrate the operation of the $Taint\_Algorithm$. The source code is given in Figure 3. The $Taint\_Algorithm$ takes as input the control flow graph $G = (V, E, r)$ where :

- $V = \{BB(1), BB(2), BB(3), BB(4)\}$
- $E = \{(BB(1), BB(2)), (BB(1), BB(3)), (BB(2), BB(4)), (BB(3), BB(4))\}$
- $r = \{BB(1)\}$

The $Dependency\_Algorithm$ checks the dependency of the blocks in the control flow graph. It generates a $Dependency\_List = \{(BB(1), BB(2)); (BB(1), BB(3))\}$. As, $BB(2)$ depends on $BB(1)$ and $BB(3)$ depends on $BB(1)$, the $Taint\_Algorithm$ sets $context\_taint$ of $BB(2)$ and $context\_taint$ of $BB(3)$ to condition taint in $BB(1)$. If $x = true$, the first branch is executed but the second is not. The first rule is used to taint modified variable y in $BB(2)$: $Taint(y) = ContextTaint \oplus Taint(explicitflowstatement)$. The second rule is used to taint the variable y in $BB(3) : Taint(y) = ContextTaint \oplus Taint(y)$. So, all variables that depend on the condition will be tainted and stored in $Tainted\_Variables\_List$ whether the branch is taken or not and we do not have an under-tainting problem.

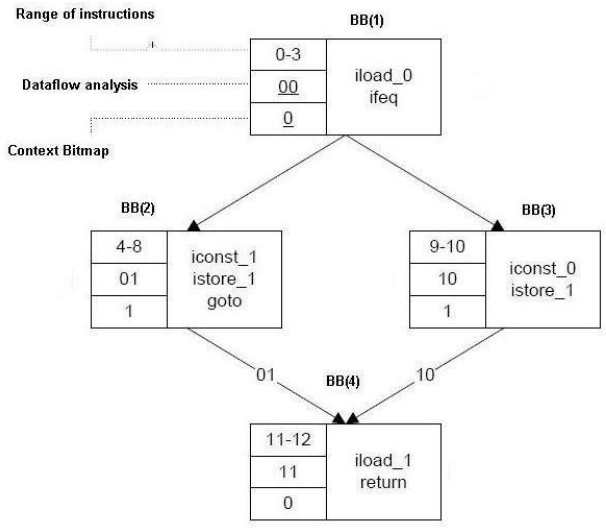

Figure 5. Control flow graph corresponding to the example given in Figure 3.

D. Properties of the algorithm

First, we prove the correctness of the $Taint\_Algorithm$ and then we prove its completeness.

1) Correctness: We want to prove the correctness of the $Taint\_Algorithm$. Let us assume that the control flow graph is correct [14]. The proof consists of three steps: first prove that $Dependency\_Algorithm$ is correct, then prove that $Set\_Context\_Taint$ is correct, and finally prove that $Taint\_Assigned\_Variable$ is correct. Each step relies on the result from the previous step.

Correctness proof for Dependency_Algorithm

The $Dependency\_Algorithm$ is defined by Ferrante *et al.* [11] to determine dependency of blocks in the graph. This algorithm takes as input the post-dominator tree for an augmented control flow graph (ACFG). Ferrante *et al.* add to the control flow graph a special predicate node ENTRY that has one edge labeled 'T' going to START node and another edge labeled 'F' going to STOP node. ENTRY corresponds to whatever external condition causes the program to begin execution. The post-dominator tree of ACFG can be created using the algorithms defined in [9], [10]. These algorithms are proven to be correct.

Basic steps in the Dependency_Algorithm:

Given the post-dominator tree, Ferrante *et al.* [11] determine control dependencies as following:

- Find S,a set of all edges (A, B) in the ACFG such that B is not an ancestor of A in the post-dominator tree (i.e., B does not postdominate A).
- For each edge (A, B) in S, find L, the least common ancestor of A and B in the post-dominator tree.

  CLAIM: Either L is A or L is the parent of A in the post-dominator tree.

  Ferrante *et al.* consider these two cases for L, and show that one method of marking the post-dominator tree with the appropriate control dependencies accommodates both cases.
  - Case 1. L = parent of A. All nodes in the post-dominator tree on the path from L to B, including B but not L, should be made control dependent on A.
  - Case 2. $L = A$. All nodes in the post-dominator tree on the path from A to B, including A and B, should be made control dependent on A.
- Given (A, B) in S and its corresponding L, the algorithm given by Ferrante *et al.* traverses backwards from B in the post-dominator tree until they reach L and mark all nodes visited; mark L only if $L = A$.
- Statements representing all marked nodes are control dependent on A with the label that is on edge (A, B).

They prove that the correctness of the construction follows directly from the definition of control dependency (see section III).

Referring back to this definition, for any node M on the path in the post-dominator tree from (but not including) L to B, (1) there is a path from A to M in the control flow graph that consists of nodes post-dominated by M, and (2) A is not post-dominated by M. Condition (1) is true because the edge (A, B) gives us a path to B, and B is post-dominated by M. Condition (2) is true because A is either L, in which case it post-dominates M, or A is a child of L not on the path from L to B.

We can therefore conclude that $Dependency\_Algorithm$ is correct.

Correctness proof for $Set\_Context\_Taint$

We include the taint of the condition in the context taint of the dependent blocks. As the condition taint is valid thus the inclusion operation is valid. We can conclude that $Set\_Context\_Taint$ is correct.

Correctness proof for $Taint\_Assigned\_Variable$

We use the two propagation rules to taint variables to which a value is assigned. We proved the completeness of the two propagation rules in section V-A, thus we can conclude that $Taint\_Assigned\_Variable$ is complete. Therefore, we can conclude the completeness of the $Taint\_Algorithm$.

2) Completeness: Let us assume that the control flow graph is complete (see Definition 2). To prove the completeness of the $Taint\_Algorithm$, we will prove the completeness of $Dependency\_Algorithm$ and $Taint\_Assigned\_Variable$.

The $Dependency\_Algorithm$ takes as input the post-dominator tree of the control flow graph. The post-dominator tree can be constructed using the complete algorithm defined in [10]. The $Dependency\_Algorithm$ is based on the set of the least common ancestor (L) of A and B in the post-dominator tree for each edge (A, B) in S. According to the value of L, Ferrante *et al.* define two cases to determine the control dependency. To prove the completeness of the $Dependency\_Algorithm$, we show that Ferrante *et al.* prove that there does not exist another value of L (either A's parent or A itself) to consider.

Proof: Let us assume that X is the parent of A in the post-dominator tree. So, X is not B because B is not an ancestor of A in the post-dominator tree (by construction of S). Ferrante *et al.* perform a proof reductio ad absurdum to demonstrate that X post-dominates B, and suppose it does not. Thus, there would be a path from B to $STOP$ that does not contain X. But, by adding edge (A, B) to this path, a path from A to $STOP$ does not pass through X (since, by construction, X is not B). This contradicts the fact that X post-dominates A. Thus, X post-dominates B and it must be an ancestor of B in the post-dominator tree. If X, immediate post-dominator of A, post-dominates B, then the least common ancestor of A and B in the post-dominator tree must be either X or A itself. ■

As only two values of L exist, there does not exist another case to compute the control dependency. The Case 2 captures loop dependency and all other dependencies are determined according to Case 1. Thus, $Dependency\_Algorithm$ is complete.

We proved the completeness of the two propagation rules in section V-A thus we can conclude that $Taint\_Assigned\_Variable$ is complete. Therefore, we can conclude the completeness of the Taint_Algorithm.

E. Time complexity of the algorithm

The $Dependency\_Algorithm$ performs with a time of at most $O(N^2)$ where N is the number of nodes in the control flow graph. Linear time algorithm to calculate control dependencies have been proposed in [15] but no proof of correctness of this algorithm was given. For each (X, Y) examined in the $Dependency\_List$, setting context taint and tainting variables can be done in constant time $O(N)$. Thus, the $Taint\_Algorithm$ requires linear time using algorithm defined in [15] and at most $O(N^2)$ using Ferrante *et al.* algorithm.

VI. Related Work

Privacy issues on smartphones are a growing concern. Several works [16], [17], [18], [19], [20] have been proposed to control access to private data in mobile operating systems. These access control approaches do not track the flow of information and cannot prevent leakage of sensitive data. A number of researches have been proposed to prevent private information leakage by untrusted android applications. [21], [22], [23] substitute private data by fake information in the data flow. This can cause a problem and disrupt execution of applications. Enck et al. [3] implement dynamic taint analysis to track explicit flows on smartphones. AppFence [23] extends Taintdroid to implement enforcement policies. One limit of these approaches is that they cannot detect control flows because they use dynamic taint analysis. The static analysis approaches implemented in smartphones [24], [25], [26] allow detecting data leaks but they cannot capture all runtime configuration and input. We were inspired by these prior works, but we combine static and dynamic analysis to prevent sensitive information leakage by untrusted android applications. Some implementations exist in the literature to solve the under-tainting problem. BitBlaze [27] presents a novel fusion of static and dynamic taint analysis techniques to track implicit and explicit flow. DTA++ [28], based on the Bitblaze approach, presents an enhancement of dynamic taint analysis to limit the under-tainting problem. However DTA++ is evaluated only on benign applications but malicious programs in which an adversary uses implicit flows to circumvent analysis are out of scope. Trishul [29] correctly identifies implicit flow of information to detect a leak of sensitive information. Egele *et al.* [30] associate a taint label with the program counter to enhance dynamic taint-tracking technique. They identify unknown components like spyware and provide comprehensive reports on their behavior. Furthermore, these approaches do not formally give a proof to solve the under-tainting problem and are not implemented in smartphones application. Fenton [31] proposed a Data Mark Machine, an abstract model, to handle control flows. Fenton associates a security class to information and defines an interaction matrix to manipulate data in the system. He gives a formal description of his model and a proof of its correctness in terms of information flow. The Data Mark Machine is based on a runtime mechanism that does not take into account the implicit flow when the branch is not executed. This can cause an under-tainting problem. To solve this problem, Aries [32] considers that writing to a particular location within a branch is disallowed when the security class associated with that location is equal or less restrictive than the security class of program counter. The Aries approach is based only on high and low security classes. Denning [33] enhances the run time mechanism used by Fenton with a compile time mechanism to solve the under-tainting problem. Denning inserts updating instructions whether the branch is taken or not to reflect the information flow. Denning and Denning [34] gave an informal argument for the soundness of their compile time mechanism. Graa et al. [4] propose an approach based on dynamic taint analysis that propagates taint along control dependencies using static analysis to track implicit flows in embedded systems such as the Google Android operating system. They define a set of formal propagation rules to solve the under-tainting problem but they do not prove the

completness of these rules. We draw our inspiration from the Denning approach, but we define formally a set of taint propagation rules to solve the under-tainting problem and we improve the approach of Graa et al. by proving the correctness and completeness of these rules.

VII. Discussion

Cavallaro *et al.* [35] describe evasion techniques that can easily defeat dynamic information flow analysis. These evasion attacks can use control dependencies. They demonstrate that a malware writer can propagate an arbitrarily large amount of information through control dependencies. Cavallaro *et al.* see that it is necessary to reason about assignments that take place on the unexecuted program branches. We implement the same idea in our taint propagation rules. Unfortunately, this will lead to an over-tainting problem (false positives). The problem has been addressed in [28] and [36] but not solved though. Kang *et al.* [28] used a diagnosis technique to select branches that could be responsible for under-tainting and propagated taint only along these branches in order to reduce over-tainting. However a smaller amount of over tainting occurs even with DTA++, as we can see by comparing the "Optimal" and "DTA++" results in the evaluation. Bao *et al.* [36] define the concept of strict control dependencies (SCDs) and introduce its semantics. They use a static analysis to identify predicate branches that give rise to SCDs. They do not consider all control dependencies to reduce the number of false positives. Their implementation gives similar results as DTA++ in many cases, but is based on the syntax of a comparison expression. Contrariwise, DTA++ uses a more general and precise semantic-level condition, implemented using symbolic execution.

In our approach, we taint all variables in the conditional branch. This causes an over-tainting. But it provides more security because all confidential data are tainted. So, the sensitive information cannot be leaked. We are interested in solving the under tainting because the false negatives are much more dangerous than the false positives since the false negatives can lead to a false sense of security. We can try to reduce the over-tainting problem by considering expert rules.

VIII. Conclusion

Smartphones are extensively used. However, the use of untrusted android applications can provoke the leakage of private data by exploiting the under-tainting problem. In this paper, we propose a formal approach to detect control flow and to solve the under-tainting problem in android sytem. We formally specify the under-tainting problem. As a solution, we provide an algorithm based on a set of formally defined rules that describe the taint propagation. We prove the completeness of those rules and the correctness and completeness of the algorithm. In the future works, we will evaluate our approach in terms of overhead and false alarms. We will also test our approach and show that it resists to code obfuscation attacks based on control dependencies.

References

[1] UK Egham, "Gartner says worldwide mobile phone sales grew 35 percent in third quarter 2010; smartphone sales increased 96 percent," November 2010, http://www.gartner.com/newsroom/id/1466313.

[2] Tim Wilson, "Many android apps leaking private information," July 2011, http://www.informationweek.com/security/mobile/many-android-apps-leaking-private-inform/231002162.

[3] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–6.

[4] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli, "Detecting control flow in smarphones: Combining static and dynamic analyses," in *Cyberspace Safety and Security*, pp. 33–47. Springer, 2012.

[5] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[6] Frances E Allen, "Control flow analysis," in *ACM Sigplan Notices*. ACM, 1970, vol. 5, pp. 1–19.

[7] Alfred V Aho and Jeffrey D Ullman, *The theory of parsing, translation, and compiling*, Prentice-Hall, Inc., 1972.

[8] Edward S. Lowry and C. W. Medlock, "Object code optimization," *Commun. ACM*, vol. 12, no. 1, pp. 13–22, Jan. 1969.

[9] Thomas Lengauer and Robert Endre Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 1, no. 1, pp. 121–141, 1979.

[10] Loukas Georgiadis and Robert E Tarjan, "Finding dominators revisited," in *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2004, pp. 869–878.

[11] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[12] S. Hedman, *A First Course In Logic: An Introduction To Model Theory, Proof Theory, Computability, And Complexity*, Number n 9 in Oxford Texts in Logic. Oxford University Press, 2004.

[13] D.E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[14] Afshin Amighi, Pedro de Carvalho Gomes, Dilian Gurov, and Marieke Huisman, "Provably correct control-flow graphs from java programs with exceptions," 2012.

[15] Richard Johnson and Keshav Pingali, "Dependence-based program analysis," in *ACM SigPlan Notices*. ACM, 1993, vol. 28, pp. 78–89.

[16] William Enck, Machigar Ongtang, and Patrick McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 235–245.

[17] Mohammad Nauman, Sohail Khan, and Xinwen Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 328–332.

[18] Mauro Conti, Vu Nguyen, and Bruno Crispo, "Crepe: Context-related policy enforcement for android," *Information Security*, pp. 331–345, 2011.

[19] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry, "Practical and lightweight domain isolation on android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 51–62.

[20] Machigar Ongtang, Kevin Butler, and Patrick McDaniel, "Porscha: Policy oriented secure content handling in android," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 221–230.

[21] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent Freeh, "Taming information-stealing smartphone applications (on android)," *Trust and Trustworthy Computing*, pp. 93–107, 2011.

[22] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM, 2011, pp. 49–54.

[23] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 639–652.

[24] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna, "Pios: Detecting privacy leaks in ios applications," in *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[25] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[26] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster, "Scandroid: Automated security certification of android applications," *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[27] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," *Information Systems Security*, pp. 1–25, 2008.

[28] M.G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation," in *Proc. of the 18th Annual Network and Distributed System Security Symp. San Diego, CA*, 2011.

[29] S.K. Nair, P.N.D. Simpson, B. Crispo, and A.S. Tanenbaum, "A virtual machine based information flow control system for policy enforcement," *Electronic Notes in Theoretical Computer Science*, vol. 197, no. 1, pp. 3–16, 2008.

[30] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song, "Dynamic spyware analysis," in *Usenix Annual Technical Conference*, 2007.

[31] J.S. Fenton, "Memoryless subsystem," *Computer Journal*, vol. 17, no. 2, pp. 143–147, 1974.

[32] J. Brown and T.F. Knight Jr, "A minimal trusted computing base for dynamically ensuring secure information flow," *Project Aries TM-015 (November 2001)*, 2001.

[33] D.E.R. Denning, *Secure information flow in computer systems*, Ph.D. thesis, Purdue University, 1975.

[34] D.E. Denning and P.J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[35] Lorenzo Cavallaro, Prateek Saxena, and R Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 143–163. Springer, 2008.

[36] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 2010, pp. 13–24.